# A Flexible SoC and Its Methodology for Parser-Based Applications

Bertrand Le Gal, Yérom-David Bromberg, Laurent Réveillère, Jigar Solanki

# A flexible SoC and its methodology for parser-based applications

Bertrand LE GAL, CNRS IMS Laboratory, Bordeaux INP, University of Bordeaux, France
Yérom-David Bromberg, University of Rennes 1 /IRISA, France
Laurent Réveillère, LABRI, University of Bordeaux, France
Jigar Solanki, LABRI, University of Bordeaux, France

Embedded systems are being increasingly network interconnected. They require to interact with their environment through text-based protocol messages. Parsing such messages is control-dominated. The work presented in this article attempts to accelerate message parsers using a codesign-based approach. We propose a generic architecture associated to an automated design methodology that enables a SoC/SoPC system generation from high-level specifications of message protocols. Experimental results obtained on a Xilinx ML605 board show acceleration factors ranging from 4 to 11. Both static and dynamic reconfigurations of coprocessors are discussed then evaluated so as to reduce the system hardware complexity.

Categories and Subject Descriptors: Hardware [**Electronic design automation**]: Hardware-software codesign

General Terms: Codesign, Hardware accelerators, Protocol parsers, Latency performance, Reconfigurability

Additional Key Words and Phrases:

**ACM Reference Format:**
ACM Trans. Embedd. Comput. Syst. 0, 1, Article 12 (January 2014), 22 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

## 1. INTRODUCTION

Embedded systems are increasingly required to interact either altogether or separately with legacy infrastructures to provide advanced services to end-users. This kind of communication between heterogeneous entities requires a protocol to manage their interactions. Traditionally, due to their highly constrained resources, embedded systems have used non-standard, application-specific, binary protocols where message parsing and parser construction are simple [Upender and Koopman 1994]. Since the use of non-standard protocols complicates the interaction with other systems, standard text-based protocols are currently preferred. For example, the SIP protocol is massively used in sensor networks [Krishnamurthy 2006] and mobile ad-hoc networks [Banerjee et al. 2007; Stuedi et al. 2007] where both computation performance and energy are limited.

Standard text-based protocol message parsers consume up to 25% of the network application time [Wanke et al. 2007]. Message parsers are typically software implemented as Finite State Machines (FSM), using a low-level language such as C to provide efficiency. Developing such parsers is still a challenging task, particularly when considering embedded system constraints. Automatic approaches such as Gapa [Borisov et al. 2007] and Zebu [Burgy et al. 2011] have been proposed to automatically generate a C implementation of the parsing automata from a high-level specification of a protocol. To the best of our knowledge, existing methodologies focus only on software code generation and have not yet explored the use of a dedicated hardware.

In the next generation of Internet (`NgN`) billions of devices related to various application domains such as aerospace, telecommunications, health-care, automotive, home automation, etc., have to smoothly interact with each other. To concretize the NgN, it is required to parse messages efficiently while saving Central Processing Unit cycles (CPU), memory and energy, which would maximize the resource constrained device's lifetime. To this end, implementing message parsers as FSM using a dedicated hardware architecture - acting as a coprocessor unit - should be an adequate choice com-

pared to a software-based implementation. A dedicated architecture can be designed specifically to execute multiple computations in parallel in one processor clock cycle. Moreover, conditional jumps, which are massively used in FSM's software implementation, are processed in one clock cycle without pipeline break penalties. Finally, a hardware-based FSM requires a lower operating clock frequency to reach the same performance than its software counterpart, thus it consumes less energy.

Nevertheless, developing a network application based on parallel processing hardware parsers is a challenging task which requires not only a hardware design and integration expertise, but also a substantial knowledge of the protocols involved, and an understanding of low-level network programming (*e.g.* sockets). When treated separately, these issues are hard to manage which explains the difficult challenge to overcome when developing a hardware protocol message parser that addresses all of them at once. In this paper, we propose a flexible co-design based architecture able to efficiently parse message streams. Moreover, we provide a complete methodology to implement efficient network applications for all kinds of resource constrained devices as encountered in the `NgN`. As such, our approach targets both Systems on Chip (SoC) and Systems on Programmable Chip (SoPC). From a high level protocol description, it generates both the Register Transfer Level (RTL) descriptions of the hardware based network message processing components and the software codes that enable their use in a network application. Generated hardware components run as coprocessors on top of a generic hardware platform. The generated software codes include both: (i) application source code template to be completed by developers, and (ii) generated software drivers that manage the generated hardware coprocessors. The contributions of this paper are as follows:

— We have designed an efficient and flexible SoC architecture for message parsing applications handling: (a) multiple hardware coprocessors working in parallel (attached like an arithmetic unit to the processor core), and (b) the deployment of network software applications. The key innovation of our approach is to provide a definition of a complete system (processor and coprocessors) that enables the development of efficient network applications leveraging on hardware accelerated parsers without requiring a mixture of high-level hardware and software engineering skills.
— We have developed a design flow allowing: (a) RTL coprocessor generation, (b) processor core adaptation, and (c) driver code and application template generation. The design flow was developed to demonstrate the interest of the approach using real application test cases. One key improvement compared to the state of the art approaches is that software and hardware parts are considered together during the SoC generation and are thoroughly evaluated.
— We have conducted a set of experiments to evaluate system efficiency against other methodologies and architectures. Different protocols such as HTTP, RTSP, SIP, and SMTP were used to assess our approach. Results have proved the system's benefits in terms of throughput, latency and flexibility.

The remainder of this paper is structured as follows: Section II presents the related works. Section III introduces a case study that illustrates a common parser-based application development. Section IV depicts: (i) the SoPC and coprocessor architectures, (ii) the design flow used to generate and/or configure them from high-level specifications. Section V summarizes the evaluation of the proposed approach. In Section VI, the reconfiguration features handled by our system on ASIC and FPGA are discussed. Section VII eventually concludes the paper.

## 2. RELATED WORKS

Over the last decade, two main approaches were used to integrate text message parsing functionalities into software applications: (i) legacy parsers and (ii) generated

parsers. Legacy parsers are based on integrating existing source codes or precompiled libraries. Thus, leveraging on legacy parsers is the fastest way to develop existing network protocols. However, reusing such kind of parsers has certain drawbacks. Most often, network applications use only a subset of data parsed from legacy parsers. As such, network applications do not require all features provided by legacy parsers (e.g. do not require to parse all fields of incoming messages), producing low-processing performances and/or high memory footprints [Burgy et al. 2011]. To overcome this issue, new approaches have emerged to avoid the painful tasks of handwriting or modifying protocol message parsers making them compliant with real requirements of network application to be implemented [Borisov et al. 2007; Burgy et al. 2011; Stefanec and Skuliber 2011]. In fact, parsers are generated from a DSL (Domain Specific Language). This approach enables automatic code generation and comprises mainly a three-step process: (i) describing network protocol messages in a high level specification (e.g. Backus Normal Form); (ii) generating software parsers from high level specifications; (iii) providing a framework to ease the development of applications on top of generated software parsers. This process helps greatly to reduce conception time for customizing protocols compared with hand coding/altering protocols. Further, it usually improves the processing performances compared with legacy parsers, particularly if only the required network applications fields are parsed (e.g. compared to parsers that by default parse all the existing fields whatever the need of the network application). However, these approaches target only software implementation of message parsers.

Our main objective is to inherit from the approach mentioned above in order to automatically generate hardware parsers from a high level language. Therefore, efficient parsers can be provided and embedded with NgN devices so as to improve their efficiency while saving CPU, memory and energy resources.

*Handwritten hardware parser.* Developing dedicated hardware [Pasha et al. 2012] should be the most efficient solution to achieve high processing performance for embedded systems. Dedicated hardware coprocessors, for instance, are mainly used for Digital Signal Processing (DSP) applications such as video processing or digital communications that are computation intensive. For these application domains, a dedicated architecture is designed specifically to implement a processing task and take advantage of multiple computations that can be realized in parallel. Massively parallel computations allow low-clock frequency usage and/or high-throughput performances compared with general-purpose processor solutions.

Applying these approaches to protocol processing applications is challenging as they are control-dominated contrary to DSP applications. However, some research has been done on the design of specific circuits for control intensive applications. State transition computations in the parser automata can involve a large set of arithmetic and logic computations that can be evaluated in parallel. For instance, in [Lunteren et al. 2004; Dai et al. 2010], a specific architecture was proposed to speed-up the eXtensible Markup Language (XML) document parsing. Similarly, in [Moscola et al. 2008], an FPGA based regular expression language is used to parse XML based streams. These works have demonstrated that FPGAs offer a viable alternative thanks to their power efficiency as well as their higher throughput when compared with software implementation. Another set of researches has been performed to speed-up regular expression processing on FPGA devices [Sidhu and Prasanna 2001; Lin et al. 2006]. In [Mitra et al. 2007], PERL Compatible Regular Expressions (PERL RegExp) were automatically transformed into hardware coprocessors for speeding-up intrusion-detection system rules using FPGAs. However, the related works mentioned above are dedicated to either (i) a particular protocol such as XML or (ii) regular expression problems to perform pattern matching [Rafla and Gauba 2010]. In the former case, it is not pos-

sible to adapt existing XML parsers for parsing other kinds of protocols. Moreover, in these works, the interconnection and the interaction between coprocessor and processor were not considered. In the latter case, our approach is similar to PERL RegExp since the expression of an FSM is equivalent to a regular language. However, we need to express value-dependent grammars that cannot be done using RegExps. In addition, our formalism is closer to Augmented Backus-Naur Form (ABNF) which is the one used for network protocol message grammars.

*Generated hardware parser*. Hardware design requires writing complex RTL code which is error prone and can be notoriously difficult to debug. Automatic compilation of a high-level specification (e.g. C codes) to silicon has been a decades-long quest in the EDA field, with an early seminal work done in the 1980s. High Level Synthesis (HLS) tools have been developed for targeting specific applications. Industrial [Y Explorations (YXI) 2010; Xilinx 2012; Mentor ] or academic [Coussy et al. 2008; Casseau and Le Gal 2012] tools synthesize a program written in a high-level language such as C into hardware circuit architecture. Similar works based on domain-specific languages and automated flows for system generation were also realized [Lucarz et al. 2008]. These approaches efficiently generate RTL architectures for computation intensive applications. However, this does not apply for control intensive applications, such as network message parsers, since they have a limited computation parallelism and lots of conditional structures.

To improve flexibility, other works have focused on a mixed hardware and software solutions. These approaches rely on the concept of an Application-Specific Instruction-set Processor (ASIP) [Pothineni et al. 2010; Pozzi et al. 2006; Sun et al. 2004; Canis et al. 2012]. An ASIP combines an instruction-set processor (ISP) core with a set of custom hardware coprocessors to improve the speed and energy efficiency. Hardware coprocessors are often closely coupled to the processor, which allows them to directly access the processor's registers. However, the processor must stall when a coprocessor performs computation. Methodologies related to automatic ASIP design focus on instruction pattern identification [Galuzzi and Bertels 2011] from source codes or RTL automatic generation [Martin et al. 2012]. The performance benefits of these approaches are important for DSP applications where patterns (that are computation intensive) are easily extracted and accelerated.

However, all these approaches are inefficient for control-based applications which mainly explains why these applications have not been widely studied. This claim is proved by experimentation results in Section 5.6.

To the best of our knowledge, previous work on hardware management for network protocol message processing has been mainly limited to XML parsing and pattern recognition using regular expressions and does not cover network message parsing requirements. Moreover, only coprocessor design was addressed. In order to provide both time performance and software flexibility, we present in this article: (i) a codesign-based system specialized for message parsing tasks and (ii) an automated methodology for automatic coprocessor and software code generations from a protocol specification.

## 3. AN ILLUSTRATIVE CASE STUDY

To ease the development of communicating applications, many of the commonly used protocols are composed of directly readable messages. However standard text-based protocol message parsers consumes up to $25\%$ of the execution time of usual network applications. This high runtime comes from the message analysis task that extracts the relevant information to generate a software view for the application layer. Let's consider the text message provided in Fig. 1. It contains the username and its associated password.

```
USER LOGIN=UserLambda PASS=*its-p@SsW0rd! END
```

Fig. 1. Text message example

```
struct authentification_msg{
  char username[64]; // contains UserLambda after the Fig. 1 message parsing
  char password[64]; // contains *its−p@SsW0rd! after the Fig. 1 message parsing
};
```

Fig. 2. Example of software view

The message parsing task consists in extracting valuable information (e.g. login and password fields) from the data stream and filling a *software view* - such as a data structure - within the extracted information (e.g. Fig. 2). The second parser task consists in validating the syntax. For instance (i) the login can contain upper and/or lower case characters and digits but no space or special characters, (ii) the message stream starts with the USER keyword and finishes with the END keyword, (iii) the information order is checked: the LOGIN field must be provided before the PASS one.

To speed-up message parser development, automated flows were developed e.g. [Burgy et al. 2011]. They generate software implementation of parsers from a high-level specification of a message protocol.

According to the protocol specification (as for example in Fig. 3), it is possible to generates: (i) the related software implementation of the protocol parser; (ii) its related data structures (Fig. 2) and (iii) a set of specialized functions for accessing the information contained in the related data structures.

This software code set enables software developers to seamlessly integrate message-parsing capabilities in their applications. Indeed, parser related source codes are automatically generated. Consequently, developers do not have to write complex automata made of dozens to hundreds of states as shown in Section 5.

## 4. PROPOSED APPROACH

### 4.1. Introduction

A well-known approach to implement an embedded application in a SoC is to develop a fully dedicated hardware architecture. However, a dedicated ASIC or an FPGA has important drawbacks. It is a tedious and time-consuming process compared with traditional software development. Moreover, dedicated design efficiency leads to less flexibility. Therefore, these solutions are discarded from network application domain, where applications usually require flexibility and programmability. To alleviate the burden in hardware-based implementations and increase design flexibility, the codesign methodology proposes to blue split an application based on the required performances. In this section, we first present the complete SoC architecture developed to efficiently parse message streams. The proposed solution does not only focus on copro-

```
%%{
machine foo_parser;
SP = ' ';
main := 'USER' SP 'LOGIN' '=' alnum+ ^login SP 'PASS' '=' alnum+
            ^pass SP 'END' ^^;
}%%
```

Fig. 3. Specification of the message protocol for automatic software generation

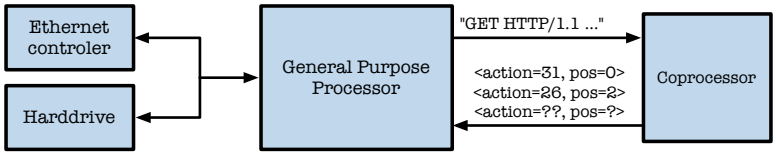


Fig. 4. Proposed approach for protocol parsing application speed-up, composed of a General Purpose Processor and a hardware coprocessor

cessor design such as related works but also handles its related software, hardware and communications issues. Then, we introduce a design flow that enables programmers to take advantage of the proposed SoC architecture.

The message parsing task consists mainly in two major parts: (i) the automata implementation (transition evaluation and next state computation) and (ii) the callback functions invoked when the automata detects a pattern or an error in the analyzed data stream. A profiling evaluation of the software parsers used in our experiments on network captured data streams shows that over $90\%$ of the execution time is spent on the automata processing. Indeed, callback functions execution time (executed once, when patterns are detected, to fill the software view) is negligible. According to these profiling results, we decided to speed-up the message parsing execution using a hardware-based implementation for the automata processing. Callback functions are left in software for flexibility reasons.

An overview of the proposed sharing out is provided in Fig. 4. The General Purpose Processor (GPP) which executes the application, provides the data stream to the coprocessor. A set of instructions, dedicated to coprocessor control and data exchange, has been integrated into the processor core. A list of the new instructions is provided in Table I.

The coprocessor analyzes the message characters and returns the processing results to the processor. The results are provided as a set (callback function identifier, stream position). The processor uses this information set to fill the software view. It invokes the callback function with the position of the character that generates this action.

A description of the different hardware and software layers involved in the proposed system is provided in Fig. 5. Software applications requiring message-parsing capabilities do not directly access neither the coprocessors, nor the low-level API. Processing executions over coprocessors are managed by the Hardware Abstraction Layer (HAL). This layer focuses on identification of coprocessors and parallel processing management (sharing hardware units between multiple threads). Moreover, this approach provides a seamless access to the protocol parsers independently from its implementation, its connectivity and the system context. The driver layer abstracts the coprocessor access API through low-level functions.

Table I. Instruction set added to the processor core to manage parsing coprocessors

| Instruction | Description |
|---|---|
| *void px_reset*(void) | Reset the coprocessor state and internal character counter. |
| *void px_send1(char)* | Send one character to the coprocessor (for parsing). |
| void px_send4(char[4]) | Send 4 characters to the coprocessor. It enables 32-bit bus usage to speed-up transfer. |
| bool px_isRunning(void) | Tells if the coprocessor is currently processing message data. |
| bool px_isIdle(void) | Tells if the processor has finished processing message data. |
| bool px_hasResults(void) | Tells if there exists a waiting processing results in the coprocessor output FIFO. |
| int px_getResults(void) | Read a processing result (couple: action, position) from the coprocessor output FIFO. |
| int px_Identifier(void) | Get the parser identifier. It indicates the protocol managed by the hardware module. |
| int px_saveContext(void) | Save the coprocessor context to enable the coprocessor usage in multi-threaded mode. |
| void px_loadContext(int) | Load coprocessor execution context to resume a stalled data stream parsing. |

### 4.2. Targeted ASIP-based architecture

One of the major bottleneck in co-design based systems is the communication between the processor and its accelerators. There are two main approaches for coprocessor coupling in a codesign-based system: (i) a closely coupled approach that integrates the coprocessor directly in the processor datapath and (ii) a generic approach that interconnects the coprocessor to the processor core using a shared peripheral bus. The first approach provides low latency data transfer times. However, it requires processor instruction set modifications, which often requires that the processor stalls while a coprocessor performs computation. The second approach is more generic as: (i) it does not require processor core modification and (ii) the coprocessor is seen as a peripheral. In this context, the processor does not stall during coprocessor computations and data transfer can be managed by a Direct Memory Access (DMA) engine. This solution is not drawback free. First, communications between the processor and coprocessors become slower due to the shared bus arbiter. Second, the communications are more complex to set up, e.g. DMA transaction configuration or management (coprocessor output data set has *a priori* an unknown length). Finally, the development and verification times are longer due to the system complexity. In the system we have proposed, we took the best of both approaches. Parsing coprocessors are closely coupled with the processor as shown in Fig. 6. They are integrated in the datapath like an arithmetic logic unit (ALU). This tight coupling allows very fast coprocessors accesses. To avoid the processor to stall while a coprocessor performs computation, the coprocessors are linked to the processor core using decoupling FIFOs (for data channels only). This architecture enables: (i) concurrent execution of computations on the coprocessors and on the processor core and (ii) different working frequency between the processor core and its coprocessors. To manage parallel execution of both units, synchronization instructions were added to the processor instruction set. This implementation looks like an extension of Xilinx's Fast Simplex Link approach for the Microblaze processor. However, in the proposed implementation, the control and the status signals are managed in an identical way.

The GPP instruction set has been extended to handle the coprocessors. To minimize the processing latency of incoming messages, the system may switch from one message stream to another. To this end, new instructions such as `saveContext` and `loadContext` have been added to save and/or load coprocessor internal states to enable interleaving of message processing depending on data availability.

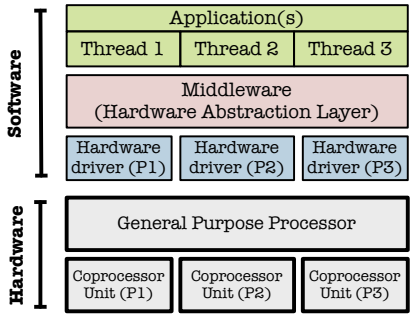


Fig. 5. Software and hardware layers involved in the proposed system

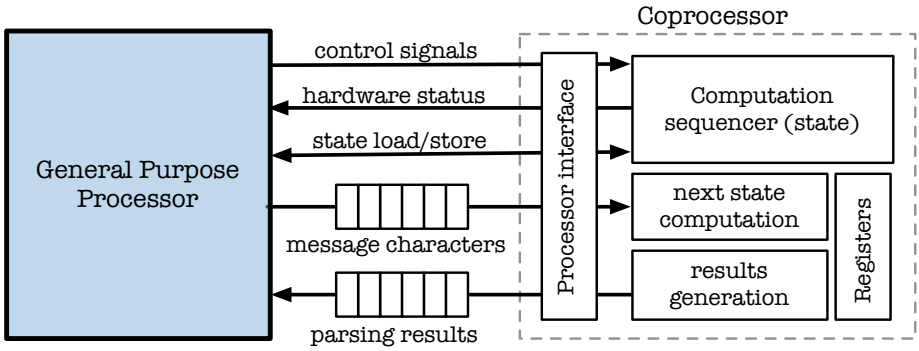


Fig. 6. Processor coupling with the protocol message parsers

### 4.3. Hardware coprocessors

Contrary to related works, e.g. on XML parser, the interconnection between the coprocessors and the processor has been taken into account. Fig. 7 presents both the interconnection links existing among the coprocessors, the processor core, and the internal coprocessor architecture. Fig. 7 provides also the width of the signals (colors are provided only to ease the figure understanding). Automaton implementation in the coprocessor is based on the commonly used Mealy FSM implementation. The message parsing specification is first transformed into a Deterministic Finite Automaton (DFA) model that implements the parsing task. States dedicated to I/O synchronization and advanced features (like context load/save) are then inserted into the model. Finally, the RTL architecture is generated.

The architecture used to implement the automaton was designed to provide timing efficiency (in terms of character processing throughput). Next state computation logic is scaled (number of arithmetical and logical resources) to enable the evaluation of the state transitions in a single clock cycle. This approach limits the processing time for a message character to from one to two clock cycles (one clock cycle when no action is triggered, two otherwise). Due to this allocation, the character processing time is independent of the number of transitions per state. In terms of hardware complexity, the hardware resources used to implement the automata depends on the maximum number of transitions per state and their computation complexities[1].

Additional resources are finally added to the coprocessor architecture to enable context loading/saving. These resources transfer the couple (state, stream position) from/to a 32-bit data that can be saved/loaded by the processor into one-clock cycle. This approach enables very fast context switching in multi-threaded applications when the overall thread requires the same coprocessor. Note that context switching requires first that the coprocessor be in idle state (input FIFO processing is finished), and second, that software layer has/will retrieve results available in the output FIFO.

The data stream transfer can be 8 or 32-bit wide at the processor output. Depending on the amount of data transferred to the coprocessor, the serializer module inserts sequentially one or four data in the FIFO resource that is 8b width. The amount of data processed by the serializer module depends on the instruction used in the software code. The fastest approach (4 characters per clock cycle) speeds-up the overall parsing

[1]Selection of other automaton models like multi-symbol automatons or other hardware implementations is still possible to achieve higher throughput or lower hardware complexity. However, such techniques were not evaluated in the current study because they were out of the objectives.

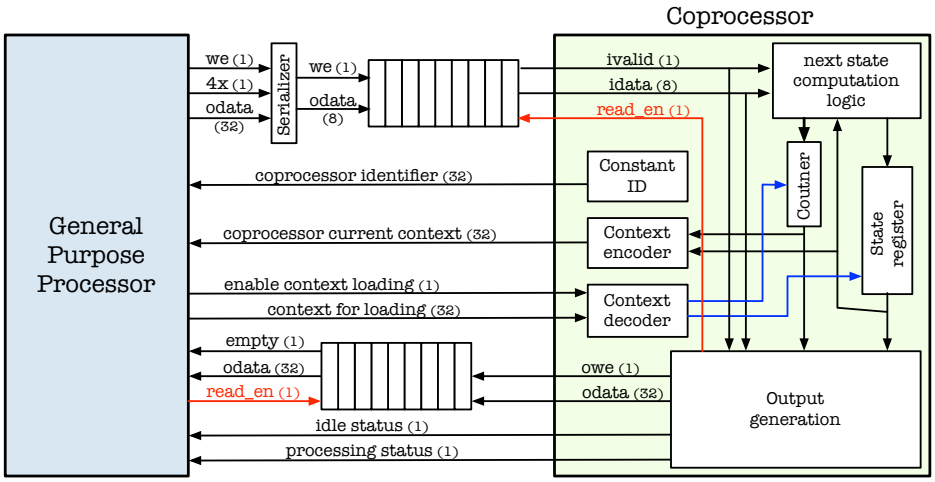


Fig. 7. Detailed view of the system architecture (numbers indicates signal width)

time. This approach hides the latency required by the processor to access character streams in its main memory and to transfer them to the coprocessor: a 32-bit data loading access provides enough characters for the coprocessor for $4$ up to $8$ clock cycles.

Result transfers from the coprocessor to the GPP are done using a 32b interface. The set (action, character position in the stream) is concatenated into a 32b word for fast processor access. Character position counter in the coprocessor was fixed in the $[0, 2^{16} - 1]$ range. The HAL is in charge of managing the counter value translation in case of longer streams. Action value was also fixed in range $[0, 2^{16} - 1]$ to ease the development so that the HAL processes the information in the same way. It is possible to adapt the action and the counter value widths according to the real coprocessor requirements.

The context saving and loading interfaces are also 32b width. State information and internal counter values that are both 16b wide are merged to fulfill the 32b data width. Using this approach, context switching can be performed within two consecutive processor clock cycles (one cycle to save the context value and the following one to restore a previously saved context). To obtain the best working frequency / hardware complexity trade-off, the state encoding format (one hot, gray, etc.) is selected by the synthesis tool. Depending on the encoding format used by the logical synthesis tool (e.g. one hot encoding format), it becomes necessary to translate the state value back on 16b (for context save and restore functions). This action and its opposite are performed respectively by the context encoder and context decoder modules shown in Fig. 7.

### 4.4. Hardware Abstraction Layer (HAL)

To process network messages, an application must register a callback function, and gives both the input socket and the protocol to be used to the HAL. The HAL manages registered applications by reading data on input streams as they are received and sending them to the corresponding coprocessor. The HAL reads then parsing results from the output interface of the coprocessor. When a message element parsing is completed, the HAL executes the ad-hoc code in order to make the value accessible by the application.

The HAL layer provides at least three main features. Firstly, it detects and starts automatically the coprocessor when the system starts. It identifies the protocol handled

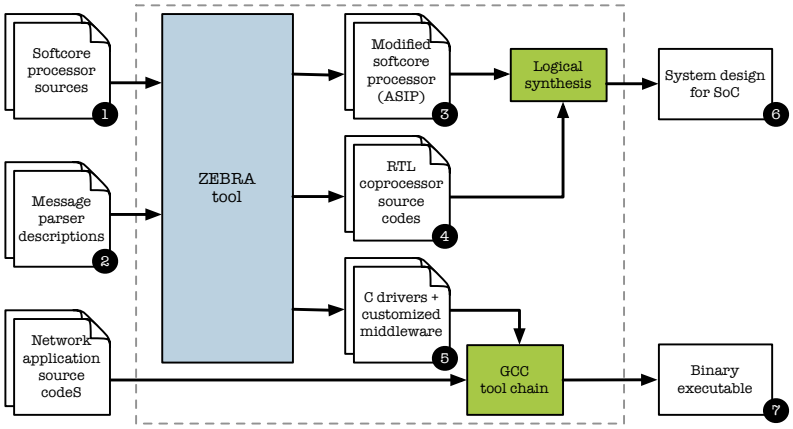


Fig. 8. Proposed design methodology for generation and configuration the system according to message parser specifications.

by each coprocessor. Secondly, it manages the coprocessor accesses for the different threads running on the processor. This service includes the context saving and restoring. Finally, it provides to the application an easy and elegant way to use coprocessor units (identical to the message processing in software).

To increase the usage rates of coprocessors between several tasks, the HAL adequately saves and restores the parser state when required. This switching on context of the hardware parsing units is very efficient and requires $8$ to $10$ clock cycles when the time of context reading/writing in the processor memory is taken into account.

### 4.5. Design methodology

Designing a coprocessor unit from protocol specifications, then integrating it and finally writing software code are long and complex tasks. Moreover, they require deep knowledge of hardware design, software development and network protocols. To demonstrate the efficiency of the proposed architecture on real use cases and to enable the development of efficient dedicated systems, an automated flow was elaborated. This methodology removes developers from the complex process of developing and using a coprocessor in a software application. Proposed design methodology is summarized in Fig. 8. Its input set is composed of:

— Specification(s) of the message protocol(s) (Fig. 8 ❷). A high-level specification language (close to the BNF notation) is used to describe text-based message formats.
— The RTL description of the softcore processor (Fig. 8 ❶). This processor core that will be interconnected with the coprocessors has to be automatically modified (e.g. creation of the interconnections to the hardware coprocessors).

According to these dataset, the proposed methodology enables to:

— Generate the RTL description(s) of the coprocessor(s) that implement the message parsing specifications (Fig. 8 ❹).
— Modify the system RTL description to instantiate and interconnect the generated coprocessors to the softcore processor (Fig. 8 ❸).
— Generate the driver codes that provide the low-level access, configuration and control functions for the generated coprocessors (Fig. 8 ❺).
— Configure the hardware abstraction layer source codes according to the number and the type of the generated coprocessors.
— Generate the associated C code tailored to application needs. This code provides standard interfaces and callback functions to enable coprocessor usage at application

```
Request = Request_Line
                (( general_header
                 | request_header
                 | entity_header ) CRLF)*
                CRLF
                message_body? {clen} ❻;
Request_Line   = Method SP Request_URI ^uri ❷ SP
                 HTTP_Version CRLF;
Request_URI    = '*' | absoluteURI | abs_path | authority;
entity_header  = Allow
               | Content_Length
               | ...
Content_Length = 'Content-Length: ' digit+ ^clen as uint32 ❶;
```

Fig. 9. Excerpt of Zebra specification for HTTP

level. The number of callback functions and their associated processing tasks are generated according to the protocol specifications.

A set of logical synthesis and software compilation scripts finish the integration process on the hardware device selected by the software developer (Fig. 8 ❻, ❼).

### 4.6. The Zebra language

Protocol specifications (Fig. 8 ❷) are provided in the Zebra language. It is based on the ABNF notation used in RFCs to specify the syntax of protocol messages to ease its adoption by network application developers. Once a basic Zebra specification is created, the developer can further annotate it according to application-specific requirements. Fig. 9 shows an excerpt of the Zebra specification for the HTTP protocol as defined in RFC 2616. Annotations define the message view available for the application, by indicating the message elements that this view should include. These annotations drive the generation of the data structure containing the message elements. For example, three message elements are annotated in Fig. 9. To make an element available, the programmer has to annotate it only with the ^ symbol and the name of a field in the generated data structure that should store the element's value. For instance, in Fig. 9, the Zebra programmer indicates that the application requires the Uniform Resource Identifier (URI) of the request line (❷). Hence, the data structure representing the message will contain one string field: `uri`.

Besides tagging message elements that will be available to the application, annotations impose type constraints on these elements. This can be specified using the notation `as` followed by the name of the desired type. For example, in Figure 9, the `Content-Length` field value (❶) is specified to represent an unsigned integer of 32 bits (`uint32`). A type constraint enables representing an element as a type other than string. The use of both kinds of annotations allows the generated data structure to be tailored to the requirements of the application logic. This simplifies the application logic's access to the message elements.

In our experience in exploring RFCs, the ABNF specification does not completely define the message structure. Indeed, further constraints are explained in the accompanying text. For example, the RFC of HTTP indicates that the length an HTTP message body depends on the `Content-Length` field value. To express this constraint, the developer has to annotate only the variable-length field `message-body` (❻) with the name of the field, between curly brackets, that defines its size (*i.e.*, `clen`). Note that such fields must be typed as an integer.

Finally, the Zebra compiler generates[2] a coprocessor tailored to the application needs according to the provided annotations, and associated C code to drive it. The coprocessor corresponds to an FSM where some transitions indicate the start or the end of message elements annotated in the Zebra specification. Thus, when such transitions are fired, the coprocessor writes into its output interface the current position of the consumed data and an action identifier. This latter specifies the name of the message element being parsed and whether it is the start or the end. This information is then used by the Zebra middleware to execute the corresponding generated callback C code.

## 5. PERFORMANCE EVALUATION

### 5.1. Introduction to experiments

In order to evaluate the performance improvement achieved by the presented system compared to fully software based ones, a set of commonly used network protocols has been selected. This includes Zebra specifications for four of the most ubiquitous protocols on the Internet (HTTP, SMTP, SIP and RTSP) which were developed (adapted from works on real software gateways presented in [Bromberg et al. 2009]). For each of them, we have used the Zebra compiler framework to automatically generate the VHDL architecture of the coprocessors and the associated software layers.

To highlight the complexity of the required automaton to implement the selected network protocols, their characteristics are provided in Table II which contains:

— the number of unfactorized transitions - It is the number of atomic transitions between states: there may exist more than one transition from state $x$ to state $y$ when the conditions $t_{(x,y)}$ are different;
— the number of factorized transitions - It is the number of complex transitions between states: there exists only one transition from state $x$ to state $y$ disregarding the condition complexity.

Note that specifications used in our experiments extract only the most commonly used subset of the protocol tokens as usually performed in network applications such as those developed for monitoring network traffic, computing statistics, billing or enabling NIDS based filters, or network protocol gateways [Bromberg and Issarny 2005; Bromberg et al. 2009; Burgy et al. 2011]. For instance, our HTTP-based application records $4$ fields for each request and response (date, host, uri for requests, status code for responses, and body). Although our HTTP specification extracts only four fields, it still consists of $55$ rules and $300$ different tokens. The number of specification lines that were required to specify the message protocol in the BNF like format were $97$, $87$, $128$ and $80$ for respectively HTTP, RTSP, SIP and SMTP protocols.

### 5.2. Hardware demonstrator on FPGA target

A prototyping system was developed targeting a Xilinx ML605 development board. The selected softcore processor was the LEON-3 [Gaisler Research 2010]. This processor core has been successfully implemented on different FPGA devices and ASIC

[2]The current compiler release relies on the use of regular expressions to implement the protocol parsers.

Table II. Complexity of the message parsing automata

| Specification | #states | Unfactorized transitions | | | Factorized transitions | | |
|---|---|---|---|---|---|---|---|
| | | #trans | Avg (trans/state) | Max (trans/state) | #trans | Avg (trans/state) | Max (trans/state) |
| HTTP parser | 66 | 3922 | 59.4 | 98 | 209 | 3.2 | 7 |
| RTSP parser | 56 | 2722 | 48.6 | 84 | 150 | 2.7 | 5 |
| SIP parser | 284 | 17451 | 61.4 | 119 | 1790 | 6.3 | 19 |
| SMTP parser | 333 | 6974 | 20.9 | 86 | 1083 | 3.3 | 7 |

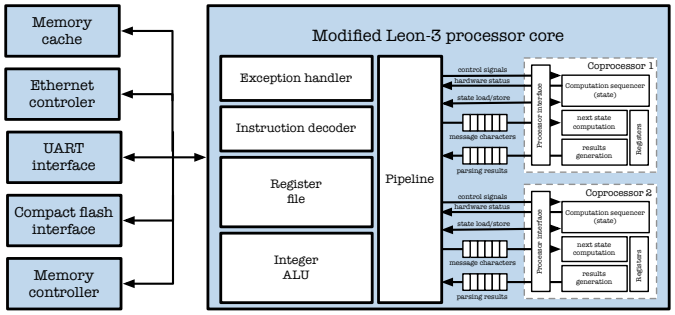


Fig. 10. Overview of the implemented system based on LEON-3 processor core (target: Xilinx ML605 development board - Virtex-6 device)

technologies. LEON-3 is fully compliant with the SPARC v8 instruction-set specification [SPARC International Inc. 1999]. The processor choice has been done based on two arguments (a) it has open-source VHDL code which allows core modifications and (b) its peripherals are fully compatible with Xilinx ML605 board. However, it is important to notice that the proposed approach is not processor core locked, other softcores may be used e.g. Plasma [Rhoads 2009], Amber [Amber Open Source Project 2013], or Altera Nios II.

The LEON-3 instruction-set has been extended to manage the generated coprocessors. The control pipeline of the processor core has also been extended to manage data exchange, context load/storing, etc. Moreover, coprocessor links has been included in the processor core like the Arithmetic and Logical Unit (ALU) to enable one clock cycle data transfer from/to the coprocessors. According to these hardware modifications, the GCC 4.4.1 toolchain has been updated to handle these new processor capabilities.

An overview of the implemented SoC is provided in Fig. 10. The complete system is composed of the LEON-3 processor core, the coprocessors and some peripherals. Included peripherals enable external DDR memory management through instruction and data caches, Ethernet communications and UART message debugging.

This system has been implemented on a ML605 development board from Xilinx. The FPGA device integrated on the board is a Xilinx Virtex-6 (XC6VLX240T-1FFG1156). Xilinx ISE 14.6 toolchain was used to generate bitstream from VHDL descriptions. Synthesis scripts and processor configurations are recovered from default LEON-3 package. The SoPC system was prototyped and evaluated using a $75$ MHz clock frequency constraint. The $75$ MHz clock corresponds to the maximum frequency that the LEON-3 processor system (version 1.3.7 b4144) can reach when synthesized by the Xilinx ISE 14.6 toolchain. The clock frequency was limited by the DDR3 controller.

Table III provides a resource breakdown of the system which consists of the ASIP core, its $4$ coprocessors and the peripherals e.g. DDR3 and Ethernet controller, memory cache, etc. The ASIP hardware estimation provided below includes the coprocessor costs.

In terms of Virtex-6 Slices, the coprocessor complexity varies from $6\%$ up to $20\%$ of the full processor. In this implementation the cost of the coprocessors represents $41\%$ of the complete system slice complexity[3]. The coprocessor complexity depends on the protocol automata complexity (number of states, transition computation complexity). Furthermore, logical synthesis results (after Place and Route (PaR)) show that

[3]However, it is important to notice that these percentages take into account only the Virtex-6 slice resources. Taking into account memory resources (RAM18k) would reduce the relative cost of the coprocessors (this is demonstrated in ASIC experimentation).

Table III. Hardware complexity - Xilinx ISE 14.6 (post-MAP and PaR under 75 MHz constraint) - Virtex-6 (ML605) - Percentages indicate device occupancy.

| | Slices | LUTs | REGs | RAM18k | Max freq. |
|---|---|---|---|---|---|
| RTSP | 298 (1%) | 812 (1%) | 208 (< 1%) | 0 (0%) | 150 MHz |
| SMTP | 703 (2%) | 2413 (1%) | 217 (< 1%) | 0 (0%) | 122 MHz |
| SIP | 1293 (3%) | 4330 (3%) | 213 (< 1%) | 0 (0%) | 123 MHz |
| HTTP | 381 (1%) | 1178 (1%) | 206 (< 1%) | 0 (0%) | 149 MHz |
| Total SoC | 13959 (37%) | 34107 (23%) | 15158 (5%) | 42 (5%) | 75 MHz |

Table IV. Characteristic of text-based messages

| | #messages | Message length (in chars.) | | | |
|---|---|---|---|---|---|
| | | Min. | Max. | Avg. | Med. |
| HTTP | 515 | 330 | 779 | 532 | 557 |
| RTSP | 205 | 56 | 210 | 135 | 151 |
| SIP | 62 | 274 | 1357 | 627 | 582 |
| SMTP | 9378 | 6 | 1003 | 118 | 27 |

the maximum coprocessors' operating clock frequency is systematically higher than LEON-3 system one. Thus, the overall operating clock frequency of the system is not slowed down by any of the coprocessors involved[4]. Consequently, one way to improve the performances of message-processing tasks is to split the clock domain into two. Thus, high frequency would be applied to coprocessors, helping them to process messages faster. Low frequency would be used by the processor core as required by its constraints. This technique requires that FIFO resources work in two different clock domains. This approach has not been used in the experimentations: the LEON-3 core and its coprocessors work at 75 MHz. Three main reasons have oriented our choice: (i) working in a Globally Asynchronous Locally Synchronous (GALS) system makes development and debugging tasks harder; (ii) performance comparison understanding would be more difficult; (iii) it won't change the experimental conclusions (performance may only be improved using this approach).

### 5.3. Processing performance evaluation

A fair evaluation of the processing performance of the prototyped system was performed to estimate the coprocessor based architecture advantages. The performance evaluation of the system was carried out using a large set of real {HTTP, SMTP, SIP, RTSP} messages captured on a real network. The captured network streams provide message characteristics for the same protocol which can change from one message to another (e.g. message length). An overview of the characteristics of the captured message streams is provided in Table IV.

Timing evaluation was performed using the automated design approach. The execution time was measured on the prototype (using a dedicated clock cycle counter) for C based implementations of the parsers and was compared to coprocessor accelerated ones. C and VHDL based parsers were generated from the same protocol specifications.

*5.3.1. System evaluation at the driver level.* A first evaluation set was performed at the driver level in which the measured execution time corresponds to the computation time required for: (i) analyzing a complete message (data transfer to the coprocessor, stream parsing), (ii) filling the application view from the parsing results (callback functions are not executed).

[4]Maximum operating clock frequencies reported in Table III correspond to the maximum frequency obtained post-PaR when the different resources are synthesized without clock constraint.

Table V. Comparison of processing performances at the driver level

| Protocol | Software parsers | | Hardware parsers | | Speed-up factors | |
|---|---|---|---|---|---|---|
| | $[min, max]$ (cycles/msg) | Driver avg (cycles/char) | [min, max] (cycles/msg) | Driver avg (cycles/char) | [min, max] | avg |
| HTTP | $[11516, 21938]$ | 29.80 | $[987, 1890]$ | 2.67 | $[9.79, 12.61]$ | $\times 11.16$ |
| RTSP | $[2434, 6453]$ | 34.51 | $[517, 765]$ | 5.08 | $[4.71, 8.61]$ | $\times 7.00$ |
| SIP | $[9278, 21627]$ | 27.66 | $[981, 2122]$ | 2.59 | $[9.28, 11.61]$ | $\times 10.68$ |
| SMTP | $[141, 8422]$ | 25.5 | $[116, 2090]$ | 9.1 | $[1.5, 5.5]$ | $\times 3.7$ |

For experimental purpose, the messages were stored in the main memory of the system. Execution time results measured experimentally on board are provided in Table V which contains the number of processor clock cycles required to parse the messages. This experimentation setup shows that coprocessor usage is efficient to reduce the message parsing execution time. Parsing speed-up factor varies from $3.7$ to $11.2$ compared with the software based solution. The speed-up factor depends on message length and the number of patterns discovered. Indeed, the execution time required to initialize the driver and to manage the hardware coprocessor becomes negligible for messages with length longer than few hundreds of characters. This fact explains the lower performances achieved by RTSP and SMTP parsers that process a large set of messages shorter than $100$ characters (Table IV).

*5.3.2. System evaluation at the application level.* Speed-up factors were also measured at the application level. This evaluation takes into account the execution time required to fill the software structure with the contents identified during the parsing processing. Structure filling requires memory allocation and memory copy of the stream content into the software structure. Execution time measurements are provided in Table VI.

Speed-ups are lower than previously. This result is due to the fact that memory allocation (malloc) and memory copying (memcpy) operations that are performed to fill the software structure consume a large number of clock cycles independently from the parser implementation. However, the improvement is not negligible: execution times are reduced $3.7$ to $4.5$ times compared with the software solution.

*5.3.3. Evaluation over a macro-benchmark.* A final experimentation set has been done to perform a macro-benchmark of the platform. Application execution times were measured when the system processes multiple network streams in parallel. The test cases are those from experimentations reported in [Bromberg et al. 2009] for the front-end part of the gateway. In this experiment, 4 message streams (one for each handled protocol) were processed in parallel to stress as much as possible the processor core and its coprocessors. Message streams were injected to the network interface of the system from a laptop computer. For each protocol, the overall message set described in Table IV was transmitted using 4 network sockets. The Ethernet connection speed was set to 100 Mbps due to LEON-3 open-source restriction. The efficiency of the proposed solution (LEON-3 and its coprocessors) was compared with two software based-solutions: one executed on a single LEON-3 core and another one composed of four LEON-3

Table VI. Comparison of processing performances at the application level

| Protocol | Software parsers | | Hardware parsers | | Speed-up factor | |
|---|---|---|---|---|---|---|
| | $[min, max]$ (cycles/msg) | App avg (cycles/char) | [min, max] (cycles/msg) | App avg (cycles/char) | [min, max] | avg |
| HTTP | $[15345, 28449]$ | 39.03 | $[3566, 6946]$ | 8.76 | $[3.70, 4.78]$ | $\times 4.47$ |
| RTSP | $[4165, 9580]$ | 53.18 | $[1512, 2723]$ | 16.41 | $[2.49, 3.85]$ | $\times 3.30$ |
| SIP | $[14804, 30779]$ | 40.63 | $[3870, 8512]$ | 9.81 | $[3.13, 4.46]$ | $\times 4.13$ |
| SMTP | $[333, 9239]$ | 41.68 | $[326, 2090]$ | 23.01 | $[1.5, 5.5]$ | $\times 3.7$ |

Table VII. System hardware complexity on 65nm ASIC technology

| Component | Constrained Freq. | NAND gates | Area (%) |
|---|---|---|---|
| LEON-3 core | 400 MHz | 158100 | 88.4% |
| HTTP | 400 MHz | 3995 | 2.23% |
| RTSP | 400 MHz | 2661 | 1.15% |
| SIP | 400 MHz | 8579 | 4.80% |
| SMTP | 400 MHz | 5446 | 3.04% |
| Total | 400 MHz | 178781 | 100% |

cores[5]. The time required to process the overall message set was respectively 31.1s and 14s for the multi-threaded applications executed on the single and the quad-core systems respectively. Execution times measured to process the same set of messages on the proposed system were only 9.4s. The experimentation complexity was *limited*. However, understanding and optimizing different stacks' behavior (hardware, drivers, HAL, TCP/IP stack, etc.) became complicated which led to analysis results reduced.

### 5.4. Hardware evaluation of the system in ASIC technology

FPGAs enable rapid SoC prototyping. However, they limit the system clock frequencies. The implementation of logic functions using LUTs and signal routing structures generates long critical path delay that may not exist in ASIC implementation. In order to evaluate more precisely the effectiveness of the proposed approach, an evaluation of the system implementation on ASIC technology was realized.

Logical synthesis of the system was done using the Synopsys Design Compiler toolchain (2013.03). Standard cell library is a 65 nm library from ST Microelectronics (CORE65 LPLVT 1.00V) that is low power optimized. The clock frequency constraint used during logicial synthesis was 400 MHz (maximum operating clock frequency of the low-power memories available in our design kit to implement the processor register banks). The LEON-3 processor core and the coprocessor elements all meet the timing requirement. The hardware complexity of the processor core and its coprocessors are provided in Table VII. The hardware complexity estimations of the different parts provided by the logical synthesis toolchain is given in NAND gates[6].

Results presented in Table VII show that the hardware complexity of the coprocessor is quite low (area of the 4 coprocessor equals 11% of the circuit area). Moreover, it can be noticed that the cost of the coprocessor set is less expensive when compared to the LEON-3 processor core in ASIC technology than in FPGA one. This difference is due to two main reasons. First, in the FPGA implementation the register bank of the processor core was done using RAM blocks which skews slice-based comparison of complexity in FPGA experimentation. Second, in ASIC technology for instance a 2-input logic function consumes less resources than a 6-input function, contrarily to FPGA technology. Concerning the operating clock frequency of the processor, it is limited to 400 MHz due to the ASIC technology used. However, we can suppose that using High Voltage Threshold (HVT) standard cell libraries or 22 nm ASIC technology would increase its operating frequency and would reduce its silicon cost.

### 5.5. System comparison with industrial processor core

To highlight the efficiency of the proposed processing system, we compared its processing performances with a commonly used industrial processor core. We made comparisons with an ARM Cortex-A9 processor core, which mainly targets power-constrained embedded applications. The evaluated ARM Cortex A9 processor core is the one in-

[5]Both systems were implemented in the Xilinx ML605 board.

[6]In current 65nm ASIC technology, a NAND gate is 2.08 $\mu m^2$, thus the silicon area is 0.371 $mm^2$. The power consumption estimated by design compiler for the system is lower than 5 mW.

Table VIII. Comparison of accelerated LEON-3 performances with ARM Cortex-A9 ones.

| Component | Processing throughput at driver level (ns/char) | | | | | |
|---|---|---|---|---|---|---|
| | LEON-3 | | ARM Cortex-A9 | | | |
| | 75 MHz | 400 MHz | 204 MHz | 370 MHz | 475 MHz | 1400 MHz |
| HTTP | 36 ns | 7 ns | 127 ns | 70 ns | 55 ns | 19 ns |
| RTSP | 68 ns | 13 ns | 312 ns | 172 ns | 134 ns | 45 ns |
| SIP | 35 ns | 7 ns | 113 ns | 62 ns | 48 ns | 16 ns |
| SMTP | 121 ns | 23 ns | 105 ns | 58 ns | 45 ns | 15 ns |

tegrated in the NVIDIA development board (Carma Tegra 3). This quad-core ARM processor is implemented in a 40 nm ASIC technology from TSMC. Each CPU core has a 32 KB of L1 instruction cache and 32 KB of data cache. The four cores share a 1 MB unified L2 cache memory. The operating clock frequency of the system is 1.4 GHz when a single ARM core is activated and 1.3 GHz in multi-core configuration. ARM-based evaluations were performed at the driver level in order to discard specific LIBC optimizations. Source codes of software parsers for the four protocols were compiled using GCC 4.6.2 with $-O3$ optimization flag. Results are provided in the Table VIII. The ARM SoC used enclosed frequency scaling, thus we forced its clock frequency to different values to provide a finer performance comparison. Processing latency measures were realized using the $clock\_gettime$ function. Note that only one processor core was switched on due to the fact that driver level benchmarks are mono-threaded.

Results provided in Table VIII demonstrate that the proposed architecture is efficient in terms of processing latency and throughput. The processor core running at 75 MHz on FPGA device achieves similar performances compared to software based parsers on the ARM core running at 475 MHz (except for the SMTP protocol). The ASIC version of the system provides better processing performance than ARM core running a 1.4 GHz (except for SMTP protocol). These results comes from the fact that ARM processor cores have 8 to 11 pipeline stages that generate costly penalties. Indeed, as an FSM requires a high amount of control and jump instructions, the behavior is difficult to predict.

### 5.6. Coprocessor efficiency compared to other C-to-RTL approaches

To validate the efficiency of the coprocessor implementation, a comparison of their execution time and hardware complexity has been performed. This comparison was done with other methodologies that generate hardware coprocessors from C source codes. The main purpose of this benchmark was to compare the hardware complexity and the timing performance of the hardware coprocessors generated by the Zebra methodology.

Five high-level synthesis (HLS) tools have been evaluated: GAUT (2.4.3) [Coussy et al. 2008], LegUP (3.0) [Canis et al. 2012], Vivado HLS (2014.2) [Xilinx 2012] from Xilinx, C to Verilog (2013) [Ben-Asher et al. 2010] and eXcite (4.1c) [Y Explorations (YXI) 2010]. To achieve the evaluation of the aforementioned tools, parsers' C source codes were generated from high level protocol specifications. All C source codes require advanced $C$ language functionalities like arithmetic pointer, goto jumps, function call, etc. Due to this large subset of ANSI C features that are required, and the automaton complexity, four of the HLS tools evaluated failed to handle the generated C source code of the different parsers.

The GAUT tool does not manage at least the goto keyword. Concerning the Vivado HLS, LegUP, C to Verilog and eXcite tools, they handle the overall semantic required by C parser codes. Nevertheless, the C to Verilog tool failed to synthesize the coprocessors and the generate RTL codes. The Excite tool compiles but crashes during the coprocessor generation due to the automata complexities. Vivado generates high complexity architectures for RTSP and HTTP parsers and crashes for the latest SMTP

and SIP parsers (*out of memory* error on a 16 GB laptop computer). Consequently, in this section we only provide a comparison of the Zebra coprocessors with the ones generated using the LegUp tool. It is important to notice that generated C codes were modified by hand to help the HLS tool. Moreover, some advanced functionalities, such as context switching, were not included in the processed C descriptions.

Hardware complexity results for the RTL coprocessors are provided in Table IX. As the LegUp tool generates RTL source codes dedicated to Altera only IP cores, results are provided using the Altera Quartus II (12.1) toolchain. To provide a fair comparison for Zebra and LegUp coprocessors, Zebra coprocessors (whose RTL source code is generic) were synthesized for the same Stratix-4 device.

Coprocessors generated with the LegUp HLS tool have a higher hardware complexity compared to our approach. Indeed, the hardware resource usage is 6 to 10 times higher. This hardware complexity increases the routing issues, generating low operating clock frequency results as demonstrated by Table IX results. Moreover, the coprocessors using the LegUp tool have a lower throughput. Indeed, they require $[2.5, 4.5]$ times more clock cycles to process a message character.

These results are on account of two major facts. First, HLS tools like LegUp were developed mainly to target computation intensive (e.g. DSP) applications. Protocol message parsers targeted in this study are control-dominated applications. Second, the LegUp tool has made the choice to instantiate one hardware resource per operation found in the application source code[Canis et al. 2012]. This choice has been made to remove usage of resources such as multiplexers that are required to share operators. In our approach, we do the opposite: we share arithmetic and logical resources as much as possible to express condition computation to reduce the overall hardware complexity. This evaluation shows that the hardware designs generated by our approach in terms of hardware complexity and throughput performance is the most adequate one to implement hardware coprocessors for message parsing applications.

## 6. SYSTEM FLEXIBITY DISCUSSION

A network application may require to manage a large set of protocols depending on its set of provided services. In the previous sections, each protocol was accelerated using a dedicated coprocessor. However, this approach is not applicable when the number of protocol increases, since it involves high hardware complexity. In this section, we discuss the possible solutions reflected upon to overcome this issue. Two scenarios are discussed according to the targeted FPGA and/or ASIC technologies.

### 6.1. Flexibility solution targeting Xilinx & Altera FPGA

Both the Xilinx and the Altera FPGA families [Xilinx 2014; Altera 2012] enable FPGA to be partially reconfigured at runtime without interrupting the overall system. This partial reconfiguration is performed using a dedicated hardware component (ICAP resource for Xilinx FPGA) and a specific synthesis methodology. A partial reconfiguration provides a solution to the mentioned flexibility problem: it provides reconfigurable zones (RZ) that can be modified at runtime. Integrating such reconfigurable zones (RZ)

Table IX. Comparison of coprocessor hardware complexity and processing performances on Altera Stratix-4 FPGA (SGX180HF35C2)

| Protocol | LegUp generated coprocessors | | | | Zebra generated coprocessors | | | |
|---|---|---|---|---|---|---|---|---|
| | ALUT | REGs | Freq. (MHz) | Avg (cycle/char) | ALUT | REGs | Freq. (MHz) | Avg (cycle/char) |
| HTTP | 7771 | 1040 | 25 | 9.40 | 1021 | 466 | 240 | 2.20 |
| RTSP | 5806 | 925 | 35 | 9.00 | 894 | 504 | 225 | 2.11 |
| SIP | 36847 | 2930 | 9 | 8.34 | 3970 | 1330 | 174 | 2.11 |
| SMTP | 29894 | 2559 | 9 | 5.05 | 2827 | 1516 | 183 | 2.06 |

Table X. Information related to partial bitstreams (footprint, reconfiguration times)

| Protocol | Without data compression | | | With data compression | | |
|---|---|---|---|---|---|---|
| | Time (cycles) | Time @75MHz | Footprint (bytes) | Time (cycles) | Time @75MHz | Footprint (bytes) |
| RTSP parser | $3.7 \times 10^6$ | $49.8ms$ | 831376 | $5.3 \times 10^6$ | $71.0ms$ | 113104 |
| SMTP parser | $3.7 \times 10^6$ | $49.8ms$ | 831376 | $5.8 \times 10^6$ | $77.9ms$ | 223008 |
| SIP parser | $3.7 \times 10^6$ | $49.8ms$ | 831376 | $6.6 \times 10^6$ | $88.3ms$ | 386632 |
| HTTP parser | $3.7 \times 10^6$ | $49.8ms$ | 831376 | $5.3 \times 10^6$ | $71.1ms$ | 116520 |

is not cost free: partial bitstream data must be transfered to the ICAP component from the system to update reconfigurable zones. Inherently, such a process requires: (i) additional execution time, and (ii) additional system memory to store the bitstream data. Moreover, RZ zones must provide a higher set of resources compared to the most costly coprocessor. Xilinx considers [Xilinx 2014] that about $80\%$ of the slices in an RZ can be used. Moreover, the working frequency of the system in the RZ may be reduced by $10\%$.

To evaluate the reconfiguration costs more precisely, a demonstrator was prototyped. It is composed of a LEON-3 core and two RZ made to demonstrate and evaluate the concept. The ICAP interface is directly connected (as a custom ALU) to the LEON-3 processor to ease the debugging task. In the implemented prototype, we decided to allocate three times more slices to the RZ than the amount required by the SIP coprocessor that is the coprocessor consuming most of the resources in our system. Such a choice of design was based two arguments. First, we want to embody new coprocessors, designed after the system is synthesized. These coprocessors can have higher hardware complexity that the ones currently implemented. Second, we want to avoid the case where routing would fail. In the current study an RZ implements at most one coprocessor (protocol) at a time. The opportunity to implement more than one coprocessor simultaneously in a single RZ when enough resources are available has not been yet evaluated. However, it seems to be an interesting way to optimize resource usage.

The partial bitstreams were stored in the LEON-3 processor memory (DDRAM) to enable partial dynamic reconfiguration at runtime[7]. The memory footprints of partial bitstreams are provided in Table X. The compression approach used to reduce memory footprint is based on the Variable Length Coding (VLC) algorithm. Execution times required for hardware reconfiguration, measured on the prototype, are also provided. Once reconfigured, coprocessor performances (cycle/character) are identical to the ones measured in previous section (Table V).

It is important to notice that the RZ were oversized when compared with the existing coprocessors' real requirements. This design choice has an impact on the memory footprint and the reconfiguration times. As useless resources are included in the RZ, reconfiguring them consumes more time and memory. However, an optimized RZ size can improve performances by a factor of $2$ to $3$. Further improvement in reconfiguration times and memory footprints would be made possible through advanced optimization techniques like the ones presented in [Duhem et al. 2012; Pham et al. 2012]. These literature approaches have not been yet added to our system demonstrator due to the huge engineering effort required to integrate them. A partially reconfigurable system helps in reducing the system hardware complexity but it has drawbacks:

— Hardware reconfiguration is time consuming, therefore simultaneous processing of multiple requests requiring different protocols on a single reconfigurable zone would be inefficient. The number of reconfigurable zones must be at least equal to the number of protocols handled by the system at the same time;

[7]Another solution is to store bitstream data in system Compact flash (if any) and to load required configuration information in memory only when the reconfigurable functionality is required.

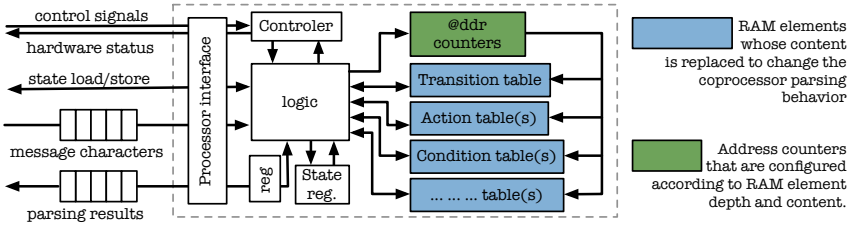


Fig. 11. Table-based coprocessor design

— Partial bitstreams required by the dynamic reconfiguration process must be stored in the system (dynamic or static memory);
— Software codes managing the hardware coprocessors and their reconfiguration are more complex which increases the complexity and implies an execution time overhead.

### 6.2. Flexibility solution for ASIC technology

Partial reconfigurable solutions are only available on Xilinx and Altera FPGA devices. In order to provide flexibility to other FPGA and ASIC based systems, a second approach has been investigated. It is based on another automata implementation which uses the State Transition Tables working with memory blocks instead of logic for transition and state computations [Sklyarov 2002; Tang et al. 2010]. An overview of the implementation architecture is provided in Fig. 11. This kind of implementation provides lower throughput performances than the previously presented one for both software and hardware targets. This performance reduction is ensued from dichotomous computations realized by the next-state evaluation function. These computations that require memory accesses need more than one cycle to evaluate the next-state according to input data and current state. The number of memories and their depths depend on the automaton to implement (# states, # transitions, condition complexities).

The main advantage of this hardware design is that the coprocessor behavior can be modified by replacing the memory blocks contents without hardware modification. This (re)configuration opportunity helps in reducing the area required in our system.

The message processing time (cycles/char) is higher compared to logic-based coprocessors: transition evaluations and next-state computations require more than one clock cycle. Compared to the (cycles/char) performance reported in Table IX, the table-based coprocessors provide lower throughputs compared to logical-based coprocessor: $\{\times 2.7, \times 3.1, \times 3.5, \times 1.6\}$ for {HTTP, RTSP, SIP, SMTP} protocols, respectively.

Reconfiguration times for a system running at $75MHz$ and memory footprints (memory content required for coprocessor reconfiguration) are provided in Table XI. These results were obtained from data with or without compression (using a Variable Length Coding algorithm). Table XI shows that the execution times required to modify the coprocessor behaviors are quite fast, proportional to memory footprint and reversely

Table XI. Coprocessor memory content reloading time and associated memory footprint

| Protocol | Without data compression | | | With data compression | | |
|---|---|---|---|---|---|---|
| | Time (cycles) | Time @75MHz | Footprint (bytes) | Time (cycles) | Time @75MHz | Footprint (bytes) |
| RTSP parser | 13802 | 0.22 ms | 1524 | 18509 | $0.30ms$ | 1307 |
| HTTP parser | 17263 | 0.28 ms | 1905 | 29012 | $0.46ms$ | 1643 |
| SIP parser | 100297 | 1.60 ms | 13196 | 136152 | $2.18ms$ | 10501 |
| SMTP parser | 70151 | 1.12 ms | 9236 | 99284 | $1.58ms$ | 7781 |

proportional to the complexity of the message-parser automatons. The hardware cost of such parser implementation is mainly composed of memory blocks: 1794 LUTs, 485 REGs. Besides, 35 BRAMs are required to handle the four protocols on Virtex-6 device. This coprocessor's architecture speed-up factor is lower than the one obtained previously. However, this architecture enables area cost reduction for systems that do not offer runtime hardware reconfiguration.

## 7. CONCLUSION

In this article, we presented an efficient and flexible SoC architecture performing text-based message analysis for network applications. The associated design flow allows an automatic generation of hardware coprocessors and software layers. Besides, it automatically configures the SoC architecture starting from high-level specification of text-based messages. Different evaluations of the approach have demonstrated that the system reduces the execution time of the message analysis task and enables parallel processing between CPU and coprocessors. Furthermore, the coprocessors' hardware complexity is lower than the one produced by traditional HLS tools. In addition, a discussion and an evaluation of the system's flexibility have shown that it is possible to support a large set of protocols with a low hardware complexity using runtime reconfiguration on both ASIC and FPGA targets. Future works will first focus on improving the design methodology algorithms then on the evolution of the SoPC architecture so that to include other platforms such as ARM cores. Finally, the evaluation of much more complex applications e.g. complete protocol gateways will be performed.

## REFERENCES


ALTERA. 2012. *Quartus II handbook version 12.1, Volume 1: design and synthesis, Chapter 3: design planning for partial reconfiguration*. Altera corporation.

AMBER OPEN SOURCE PROJECT. 2013. *Amber 2 Core Specification*.

BANERJEE, N., ACHARYA, A., AND DAS, S. K. 2007. Enabling sip-based sessions in ad hoc networks. *Wireless Netw. 13,* 4, 461–479.

BEN-ASHER, Y., MEISLER, D., AND ROTEM, N. 2010. Reducing memory constraints in modulo scheduling synthesis for fpgas. *ACM Transactions on Reconfigurable Technology and Systems 3,* 3, 15:1–15:19.

BORISOV, N., BRUMLEY, D. J., WANG, H. J., DUNAGAN, J., JOSHI, P., AND GUO, C. 2007. A generic application-level protocol analyzer and its language. In *14th Annual Network & Distributed System Security Symposium*. 14:1–14:14.

BROMBERG, Y.-D. AND ISSARNY, V. 2005. INDISS: Interoperable discovery system for networked services. In *International Conference on Middleware*. 164–183.

BROMBERG, Y.-D., RÉVEILLÈRE, L., LAWALL, J. L., AND MULLER, G. 2009. Automatic generation of network protocol gateways. In *International Conference on Middleware*. 21–41.

BURGY, L., RÉVEILLÈRE, L., LAWALL, J., AND MULLER, G. 2011. Zebu: A language-based approach for network protocol message processing. *IEEE Transactions on Software Engineering 37*, 575–591.

CANIS, A., CHOI, J., ALDHAM, M., ZHANG, V., KAMMOONA, A., CZAJKOWSKI, T., BROWN, S., AND ANDERSON, J. 2012. Legup: An open source high-level synthesis tool for fpga-based processor/accelerator systems. *ACM Transactions on Embedded Computing Systems (TECS) 1,* 1, 1–25.

CASSEAU, E. AND LE GAL, B. 2012. Design of multi-mode application-specific cores based on high-level synthesis. *Integration, the VLSI Journal 45,* 1, 9–21.

COUSSY, P., CHAVET, C., BOMEL, P., HELLER, D., SENN, E., AND MARTIN, E. 2008. *High-Level Synthesis*. Springer, Chapter GAUT: A High-Level Synthesis Tool for DSP Applications, 147–169.

DAI, Z., NI, N., AND ZHU, J. 2010. A 1 cycle-per-byte xml parsing accelerator. In *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*. FPGA'10. 199–208.

DUHEM, F., MULLER, F., AND LORENZINI, P. 2012. Reconfiguration time overhead on field programmable gate arrays: reduction and cost model. *IET Computers & Digital Techniques 6,* 2, 105–113.

GAISLER RESEARCH. 2010. *GRLIB IP Library User's Manual*.

GALUZZI, C. AND BERTELS, K. 2011. The instruction-set extension problem: A survey. *ACM Trans. Reconfigurable Technol. Syst. 4,* 2, 18:1–18:28.

KRISHNAMURTHY, S. 2006. Tinysip: Providing seamless access to sensor-based services. *Proc. 3rd Int. Conf. Mobile Ubiquitous Sys.: Netw. Serv.* 4611, 1–9.

LIN, C.-H., HUANG, C.-T., JIANG, C.-P., AND CHANG, S.-C. 2006. Optimization of regular expression pattern matching circuits on FPGA. In *Proceedings of the DATE Conference*. 12–17.

LUCARZ, C., MATTAVELLI, M., WIPLIEZ, M., ROQUIER, G., RAULET, M., JANNECK, J., MILLER, I., AND PARLOUR, D. 2008. Dataflow/actor-oriented language for the design of complex signal processing systems. In *Proceedings of the DASIP Conference*. 168–175.

LUNTEREN, J. V., ENGBERSEN, T., BOSTIAN, J., CAREY, B., AND LARSSON, C. 2004. Xml accelerator engine. In *1st Int. Workshop on High Performance XML Processing*.

MARTIN, K., WOLINSKI, C., KUCHCINSKI, K., FLOCH, A., AND CHAROT, F. 2012. Constraint Programming Approach to Reconfigurable Processor Extension Generation and Application Compilation. *ACM TRETS 5,* 2, 1–38.

MENTOR. *Catapult C Synthesis User's and Reference Manual*.

MITRA, A., NAJJAR, W., AND BHUYAN, L. 2007. Compiling PCRE to FPGA for accelerating SNORT IDS. In *Proceedings of the Symposium on Architecture for networking and communications systems*. 127–136.

MOSCOLA, J., CHO, Y. H., AND LOCKWOOD, J. W. 2008. Reconfigurable content-based router using hardware-accelerated language parser. *ACM Trans. on Design Automation of Electroni Systems 13,* 2.

PASHA, M. A., DERRIEN, S., AND SENTIEYS, O. 2012. System-level synthesis for wireless sensor node controllers: A complete design flow. *ACM TODAES 17,* 1, 1–24.

PHAM, M., BONAMY, R., PILLEMENT, S., AND CHILLET, D. 2012. Power-aware ultra-rapid reconfiguration controller. In *Proceedings of the Conference on Design and Test in Europe (DATE)*. 1373–1378.

POTHINENI, N., BRISK, P., LENNE, P., KUMAR, A., AND PAUL, K. 2010. A high-level synthesis flow for custom instruction set extensions for application-specific processors. In *ACM/IEEE Asia and South Pacific Design Automation Conference*. 707–712.

POZZI, L., ATASU, K., AND LENNE, P. 2006. Exact and approximate algorithms for the extension of embedded processor instruction sets. *IEEE Trans. on CAD of Integrated Circuits and Systems 25,* 7, 1209–1229.

RAFLA, N. AND GAUBA, I. 2010. A reconfigurable pattern matching hardware implementation using on-chip ram-based fsm. In *Proceedings of the MWSCAS Conference*. 49–52.

RHOADS, S. 2009. *Plasma - most MIPS I(TM) opcodes: Overview*.

SIDHU, R. AND PRASANNA, V. K. 2001. Fast regular expression matching using fpgas. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*. 227–238.

SKLYAROV, V. 2002. Reconfigurable models of finite state machines and their implementation in fpgas. *EUROMICRO Journal of Systems Architecture 47,* 14-15, 1043–1064.

SPARC INTERNATIONAL INC. 1999. *The SPARC Architecture Manual*.

STEFANEC, T. AND SKULIBER, I. 2011. Grammar-based SIP parser implementation with performance optimizations. In *Proceedings of the International Conference on Telecommunications*. 81–86.

STUEDI, P., BIHR, M., REMUND, A., AND ALONSO, G. 2007. Siphoc: Efficient sip middleware for ad hoc networks. In *Proceedings of the ACM International Conference on Middleware*. 60–79.

SUN, F., RAGHUNATHAN, A., RAVI, S., AND JHA, N. 2004. Custom-instruction synthesis for extensible-processor platforms. *IEEE Trans. on CAD of Integrated Circuits and Systems 23,* 7, 216–228.

TANG, Y., JIANG, J., WANG, X., WANG, Y., AND LIU, B. 2010. Cache-based scalable deep packet inspection with predictive automaton. In *Proceedings of the IEEE Global Telecommunications Conference*. 1–5.

UPENDER, B. AND KOOPMAN, P. 1994. Communications protocols for embedded systems. *ACM Transactions on Programming Languages and Systems (TOPLAS) 11,* 7, 46—58.

WANKE, S., SCHARF, M., KIESEL, S., AND WAHL., S. 2007. Measurement of the sip parsing performance in the sip express router. *Lecture Notes in Computer Science* 460, 103–110.

XILINX. 2012. *Vivado Design Suite User Guide: Synthesis*. UG901 (v2012.2).

XILINX. 2014. *Vivado Design Suite User Guide - Partial Reconfiguration (UG909)*. Xilinx corporation.

Y Explorations (YXI) 2010. *eXCite C to RTL Behavioral Synthesis 4.1(a)*. Y Explorations (YXI).

## ELECTRONIC APPENDIX

The electronic appendix for this article can be accessed in the ACM Digital Library.

Received March 2014; revised July 2014; accepted —